2008

Historical Study of the Development of Branch Predictors

Yuval Peress

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

HISTORICAL STUDY OF THE DEVELOPMENT OF BRANCH PREDICTORS

By

YUVAL PERESS

A Thesis submitted to the
Computer Science Department
in partial fulfillment of the
requirements for the degree of
Master of Science.

Degree Awarded:
Fall Semester, 2008

The members of the Committee approve the Thesis of Yuval Peress defended on September 29, 2008.

Gary Tyson
Professor Directing Thesis

David Whalley
Committee Member

Piyush Kumar
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In all areas of research, finding the correct limiting factor able to provide the largest gains can often be the critical path of the research itself. In our work, focusing on branch prediction, we attempt to discover in what ways did previous prediction research improve branch prediction, what key aspects of the program execution were used as leverage for achieving better branch prediction, and thus which trends are more likely to provide more substantial gains. Several "standard" branch predictors were implemented and tested with a wide variety of parameter permutations. These predictors include the bimodal, GAg, gshare, local, and tournament predictors. Later, other predictors were studied and briefly described with a short analysis using information gathered from the "standard" predictors. These more complex predictors include the caching, cascading look-ahead, overriding, pipelined, bi-mode, correlation-based, data correlation based, address correlation based, agree, and neural predictors. Each of these predictors have their own unique approach on which elements of the branch predictor are key to improving the prediction rate of a running benchmark. Finally, in our conclusion, we will clearly state our perspective on which elements in branch prediction have the most potential and which of the more advanced predictors covered in the related work chapter have been following our predicted trend in branch prediction development.

CHAPTER 1

INTRODUCTION

Branch predictors have come a long way over the years; from a single table indexed by a current PC, to multiple layers of tables indexed by the PC, global history, local history, and more. The final goal is the use of past events to predict the future execution path of a running program. At each step in the evolution of branch predictors we have seen new improvements in the prediction accuracy, generally at the cost of power and some space. In recent designs, such as the Alpha EV8[9][1], the miss rate of the branch predictor could indeed be the limiting factor of the processor's performance. The EV8 was designed to allow up to 16 instructions to be fetched per cycle, with the capability of making 16 branch predictions per cycle. With increasing penalties for misprediction (minimum of 14 cycles in the EV8) and a high clock frequency limiting the number of table accesses of the predictor, the EV8 processors was designed with 352 Kbits allocated for its branch predictor.

While we have been seeing improvements in prediction accuracy over the years, no research has gone into finding exactly what elements of the branch predictor design are used as leverage to generate the increased performance and finally: what has yet to be used for leverage. The immediate goal of this study is to show how different predictors leverage different features of a program's execution to achieve better prediction rates. Further, we will describe how these features can be used to classify branches and benchmarks according to their behavior with varying styles and sizes of branch predictors. The ultimate goal of the research is to create a standard in branch prediction research providing a new method to present the data collected as well as show relevance of using new statistics beyond a simple miss rate to analyze individual branch predictors.

Other studies, comparing different branch predictors, have been published[6][8][7]. In his

[1]The EV8's wide-issue deeply pipelined design is used here as an example of modern processor trends.

paper[6], J. E. Smith compares the prediction rate of early branch prediction schemes from "predict taken" to the bimodal branch predictor. The research done for this paper begins with the bimodal predictor and will conclude with a review of modern complex branch predictors. Lee and Smith published a similar paper[8] where different Branch Target Buffer designs along with prediction schemes are compared for an IBM architecture. Like our paper and many others, Lee and Smith subdivided their benchmarks into categories. Unlike our research, these categories are created by the type of application the benchmark falls under, whereas our research only sub-divides the benchmarks once a clear difference can be made within the benchmark from their individual branch instructions. In [7], different styles of a more sophisticated two level predictor implementations and configurations were compared to determine the best performing. While this paper is similar in concept, our analysis will delve into the true cause of prediction improvement or degradation using specific examples of branches and table entries. In all previous studies of different branch prediction schemes reviewed prior to starting this research, only the average prediction rate or reduction of the miss penalty of each scheme was presented as the end result of the experiment. Throughout the paper we will expose the need for a new format for data presentation for branch prediction research, exposing the flaws in the simple average miss rate across the given benchmarks. Further, we will explore specific branch instructions and show conclusively why certain branch predictors out-perform others.

To achieve all of these goals we will first cover the basics of the research, such as the simulator used, the benchmarks studied, and the peripherals of the branch predictor before examining specific predictors. We will then cover each "basic" predictor chosen as a baseline studying the internal workings of each, and the effect of the varying configuration of these specific predictors. Following the detailed study of each predictor we will compare each predictor to the next. We will then review current trends of more modern predictor development that use the selected predictors as their baseline as well as a few other predictor schemes. Ultimately, we will demonstrate that some trends in branch prediction research have been exhausted and are not likely to provide additional gains to merit focus. We will also reveal other trends in development that have been shown to work and provide a high potential. To evaluate such trends we will present a new method for reviewing branch prediction statistics as well as a few new metrics used to evaluate new predictors.

1.1 Simulation Environment

All simulations were run under the M5 simulation environment[1]. M5 provides a detailed simulation of a complete out of order CPU. For the purpose of this research, only the branch prediction aspects of M5 were modified to provide the needed statistics. For the purpose of this study benchmarks were run on M5 using the System Call Emulation mode. Such an execution allowed us to analyze the execution of a single program without interference from the kernel code or other processes running in the background.

1.2 Benchmarks

For this study we used a large number of benchmarks from MiBench, MediaBench, and a few other test programs written specifically for this study. The following is a list of the benchmarks used:

- MiBench: FFT, adpcm, basicmath, bitcount, dijkstra, ghostscript, jpeg, lame, patricia, stringsearch, and susan.
- MediaBench: epic, g721, gsm, and mpeg2.
- Other: bubblesort, quicksort, and Gale Shapley's algorithm.

CHAPTER 2

THE PERIPHERALS OF THE BRANCH PREDICTOR

Like any other element, the final branch prediction depends on more than just a single component. A final prediction at the end of the cycle is the end result of both the direction predictor and the Branch Target Buffer (BTB). Generally, the direction predictor uses a table of small state machines, normally 2 bits (4 states), called the Pattern History Table (PHT). The difference between different predictors is then usually the indexing scheme into the PHT. The BTB is a table, similar in design to a cache, storing a mapping of branch PCs and their taken target PCs. While the direction predictor is the focus of this research, this chapter will cover the peripherals of the direction predictor such as the BTB and the 2 bit state machine.

2.1 The BTB

The BTB is a data structure used in parallel to the branch predictor. It is the BTB that tells us if the current PC is a previously taken branch and what the target PC of that branch is, assuming it is taken. The BTB is often set associative and ranges in size. For our study we chose to use 1k-4k 4-way associative BTB tables.

Knowing that, on average, a branch instruction will be executed every 3-4 instructions, we can already deem the use of the BTB as a wasteful one due to the need for an access for each instruction fetched. Next, we must look at how efficiently we use the table size of the BTB. Figure 2.1 reveals just how wasteful the BTB size can be. All the benchmarks were run and data was collected on how many branches were executed between executions of the same branch thus providing the branch working set size (i.e. if one branch executes, then another branch executes followed by the original branch, we can say the working set is 2

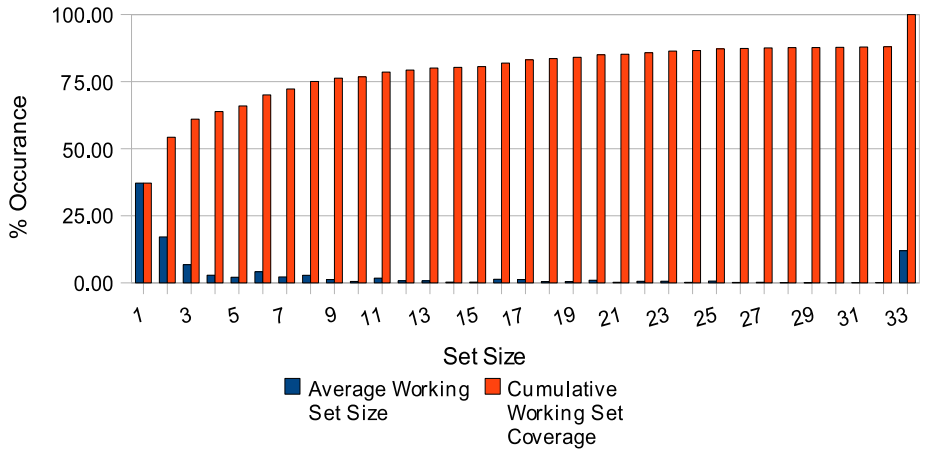


Figure 2.1: The average working set size within the BTB. The last column (33) is the overflow, meaning any working set size > 32 is placed in column 33.

branches). The data for Figure 2.1 was taken for a 2k 4-way associative BTB table. We can see that 82% of the working sets can be captured with only 16 entries of the BTB. Further, 32 entries in the BTB would yield 88% of the working sets. Unfortunately, no indexing scheme available can map branches into only 32 entries while maintaining the time constraints of a single cycle access.

2.2 The 2-Bit State Machine

A branch predictor is an architectural feature located in the fetch stage of the CPU pipeline. The purpose of the branch predictor is to first identify fetched branches; and second, to predict the result of that branch so that the next instruction can be fetched without waiting for the result of the branch to be calculated. As pipelines become deeper and the time to calculate the result of the branch increases, the penalty for missing a branch prediction or simply stalling becomes greater in respect to the number of instructions that can commit per unit time.

The prediction of a branch is generally done via a 2-bit state machine replicated and placed in a table, the PHT. The main difference between predictors then becomes the indexing method used to read and update the state machine. The state machine, described

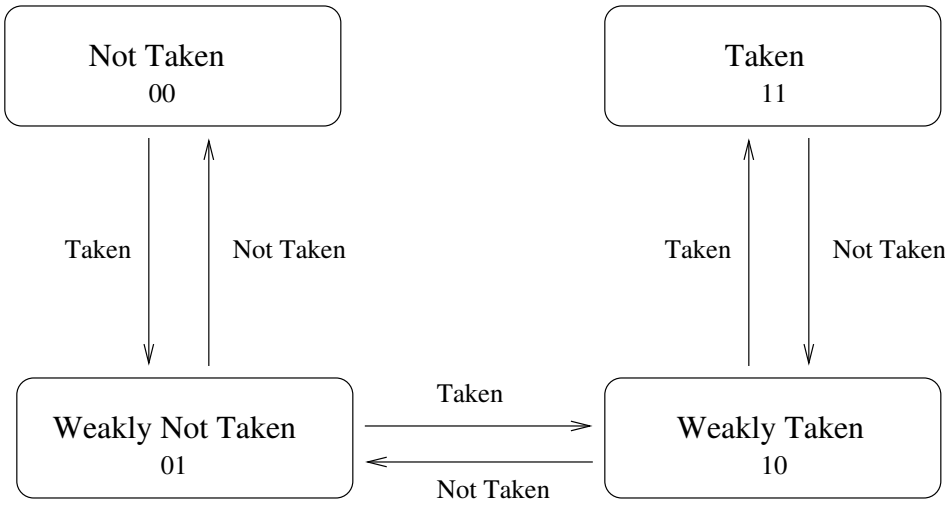


Figure 2.2: Transition diagram for 2bit predictor.

in Figure 2.2, is a saturated 2-bit counter. The counter is incremented or decremented when the branch is taken or not taken respectively. Being a saturated counter, the 2 bits will not be decremented once reaching 00 and will not be incremented once reaching 11. Timing is also key in updating the counter. Simpler predictors choose to update only when the branch instruction has been resolved, others update speculatively using the predicted path; but generally, no speculative updates are done to the saturating counter.

CHAPTER 3

CHANGING THE TABLE SIZE

The first and most obvious comparison is to compare each predictor's performance to its own performance when using a larger table. Specifically, we were looking for changes in contention for entries in the tables.

3.1 Bimodal Predictor

The bimodal predictor is the simplest predictor next to a *predict same as last* predictor. The goal of the bimodal predictor is to easily predict branches which exhibit consistently taken or not taken behavior. The bimodal predictor works as follows: a table containing n entries and having a 2 bit saturating counter per entry is indexed via the lower order $\log_2 n$ bits of the PC. The 2 bits are then used as a simple state machine/saturating counter demonstrated in Figure 2.2. At each update, the 2 bit counter will be incremented or decremented according to the true result of the branch. It is important to recognize that this predictor only makes use of local (PC specific) branch resolution, and is vulnerable to interference by other branches indexing to the same entry. This means that branches with certain patterns (TNTNTN) will always miss due to the state machine design, as well as two separate interchanging branches which index into the same saturating counter in different direction.

As can be seen in Figure 3.1 we do find the very expected pattern of lower contention in higher size tables as well as more instances of unused entries. We had found that an astounding 41.1% of the 1k bimodal prediction table was completely unused. Further, increasing the table from 1k to 8k only reduced the raw number of entries with contention by a factor of 4.

This phenomenon can possibly be explained by the distribution of branches. As can be seen in Figure 3.2a, branches rarely occur immediately after one another. Thus, leaving

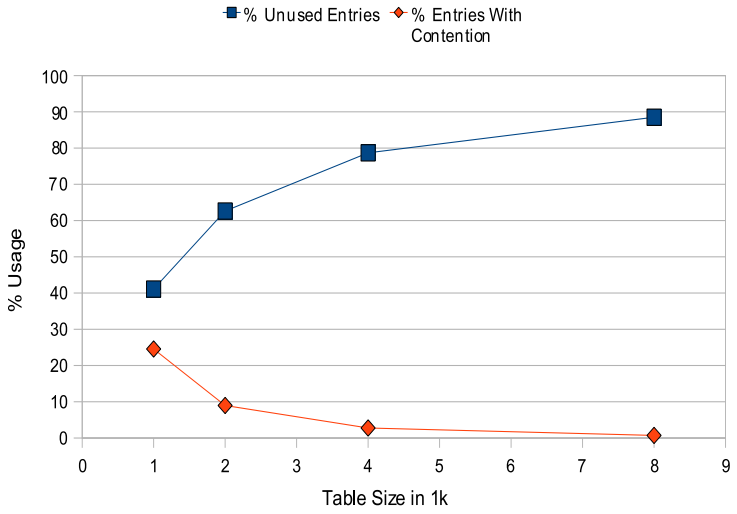
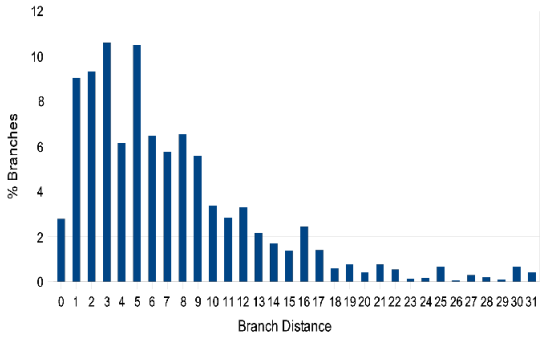


Figure 3.1: Average Bimodal Predictor Table Usage

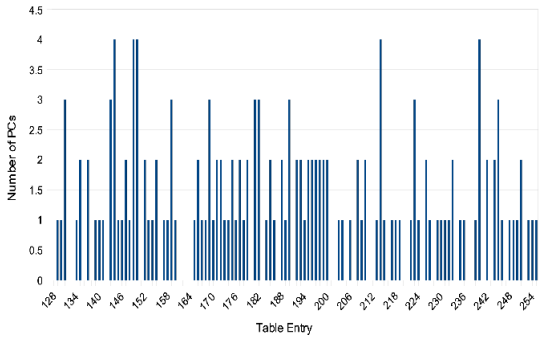
gaps in a PC indexed table, shown in Figure 3.2b, and can be described mathematically by looking at the average number of PCs accessing each table entry (1.21) and the standard deviation (1.04) for the raw number of PCs depicted in Figure 3.2b. This means that 0-2 different PCs could be accessing any given entry (within only 1 standard deviation). Further, between all benchmarks we found that we have an average number of PCs per table entry of 0.9 and a standard deviation of 0.86.

Beside being a wasteful increase in size, most of which is unused, an increase in bimodal table size hardly helps the miss rate of the 18 studied benchmarks (Table 3.1). While most benchmarks do benefit from the reduction in contention slightly, some benchmarks benefit from *constructive interference* in smaller table sizes: when two poorly predictable branches map to the same predictor entry and modify each other's prediction in such a way that the net result is better than if they were to map to different entries.

Unfortunately there are many more cases of destructive interference than constructive interference, thus driving the need for larger tables in such simple indexing schemes. To be more specific we found that the average reduction in miss rate when removing destructive interference is 31.81%, while removing constructive interference increases the miss rate by only 2.9%. Further, there were 21.6% more cases of destructive interference than constructive



(a) Branch Distribution



(b) Bimodal Table Contention Sample from benchmark: Patricia

Figure 3.2: Branch predictor usage patterns in 1k bimodal predictor.

interference[1].

Table 3.1: Bimodal Miss Rates

Table size	Avg. Miss Rate	Std. Dev.
1k	7.19%	6.27
2k	7.01%	6.31
4k	6.87%	6.31
8k	6.84%	6.32

Due to the data presented in Figure 3.2a and Figure 3.2b we decided to collect a second set of statistics for the bimodal predictor using a slightly modified indexing scheme. Since branches rarely occur in consecutive PCs (derived from Figure 3.2b) and branches are also rarely executed with no non-branch instructions between them, shown in Figure 3.2a. It would appear that simply ignoring the lowest order bit of the PC would yield a better use of the table size and perhaps provide a more even distribution unlike the one shown in Figure 3.2b. The data collected was done with table sizes half those present in Table 3.1 and is presented in Table 3.1. While the data shown reveals no improvements were made in each table size, it also confirms our hypothesis regarding the low order bit. When looking at the

[1]A further study with specific examples of both destructive and constructive interferences can be found in Appendix B and Appendix C.

specific miss rates of each benchmark and for each table size we find that the miss rate of the shifted bimodal predictor is very similar to that of the bimodal predictor doubled in size for benchmarks having strong constructive or destructive interference. For example: the standard 1k bimodal predictor achieves a 6.81% miss rate for the *lame* benchmark, while the shifted 1k bimodal predictor yields a 10.45% miss rate (much like the 10.41% miss rate of the 2k standard bimodal). In the previous example, the standard bimodal predictor's indexing scheme created a case of constructive interference in the 1k table size which was then lost when addition higher order PC bits were used in larger tables. When the shifted bimodal predictor used PC bits 1-10 instead of 0-9, it lost the constructive interference due to a differing index created by the 10th bit.

Table 3.2: Bimodal Miss Rates for the shifted PC indexing scheme.

Table size	Avg. Miss Rate	Std. Dev.
512	7.46%	6.47
1k	7.24%	6.43
2k	7.12%	6.45
4k	6.97%	6.46

3.2 GAg and Gshare

The next step in branch prediction evolution is the Global Adaptive branch predictor using one *global* pattern history register, or GAg[3]. The GAg still utilizes a table of n entries and 2 bits per entry to represent the same state machine described in Figure 2.2. The difference is in the indexing scheme: previously, we used the lower order $\log_2 n$ bits of the PC to index into the table. The new scheme uses the lower order $\log_2 n$ bits of a global history register to index into the table, thus finding correlations between previously executed branches and the current prediction. This means that the prediction depends on the previous pattern of Taken or Not Taken branches. Such an indexing scheme allows a more efficient use of the table via grouping branches with similar patterns into using the same entries in the table.

GAg, being completely global history based, seems to improve greatly for every additional bit of global history added. A pattern such as this is the result of intermixing of branches in the global history register. In other words, the longer the global history register allows

Table 3.3: Gag Miss Rates

Table size	Avg. Miss Rate	Std. Dev.
1k	7.92%	5.37
2k	7.50%	5.24
4k	7.29%	5.07
8k	6.86%	4.82
16k	6.58%	4.68
32k	6.15%	4.66
64k	5.73%	4.70
128k	5.56%	4.73

the GAg predictor to find stronger correlations between global patterns and current branch prediction. Also, given enough training time and a long enough history register, the GAg predictor is able to inefficiently capture local behavior of branches by saturating the many different PHT entries corresponding to the local branch's pattern and the permutations of the other possible bits added to the global history register by other branches in the working set. Finally, it appears that nearly all of the improvements visible in increasing the history register and table size are caused by an increase of the number of bits present in the global history register inserted by the same PC being fetched, thus acting like a local history register[2].

The next step was to create the gshare predictor[2]. The gshare is the same GAg predictor with a small alteration: to index into the prediction table we first XOR the PC of the instruction with the history register. This allows for a much more evenly spread usage of the table space than the bimodal predictor, while still indexing potentially conflicting branches into separate entries. The key reasoning behind the combination of the PC and global history is to make use of both temporal and spacial attributes of the branch, thus reducing the table size needed to achieve the same prediction rate.

Similar to other predictors, most of the benefits are derived in the first few increments of table size. Contention, in the case of gshare, is generally eliminated by the indexing scheme itself, which is greatly influenced by the global history. We find that instead the table size

[2]A more in-depth study, presented later in this section, was done on the branch PCs contributing to the global history register and miss rates for the gshare predictor.

Table 3.4: Gshare Miss Rates

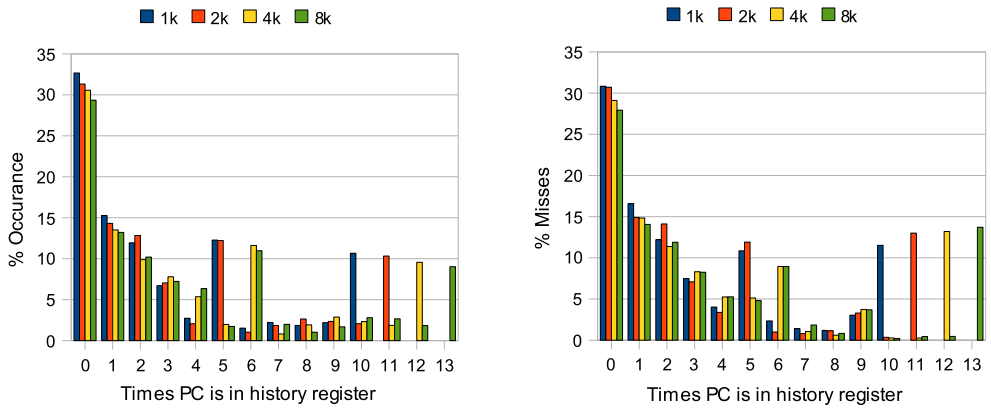
Table size	Avg. Miss Rate	Std. Dev.
1k	6.76%	5.48
2k	6.12%	5.40
4k	5.77%	5.31
8k	5.66%	5.25

increase is only useful in capturing correlations with wider branch working sets since it was found that contention in the PHT of gshare is evenly spread between all entries with a low standard deviation.

Since some branches are very easy to predict, it made sense to also look at the effect of the change in table size on the top 10 missed branches. We know that as a whole the miss rate decreases as the table size and history length increase, alternatively we will now look at which branch PCs are present in the top 10 missed branches. Two different statistics were gathered to allow a proper comparison of the 1k, 2k, 4k, and 8k gshare configurations. The first was a count of how many new branches were present in both the top 10 lists of a gshare size N and $2N$, i.e. how many branches were replaced with new branches when the table size doubled. The second was a count of how many branches shifted in the sorted top 10 missed branches list. We had found that on average there are (1.6, 1.0, 1.1)[3] new branches in the top 10; we also found that on average (4.1, 4.2, 3.4) branches shift their position in the top 10. These values, reduction in misses of the top 10 branches combined with the new branch and shifted branch count, reveal that, in general, the top missed branches are more likely to remain highly missed when increasing the table size.

At each increment of table size, the history length is increased by 1 bit. Further, the number of bits used to index into the table from the PC is also increased by 1. These facts provide for 2 patterns of behavior. The first allows for branches spaced far apart in a branch working set to provide prediction "hints" for currently fetched branches. This extends over into the same branch executed within n instructions to account for a portion of the history register of gshare, thus simulating local prediction in small working sets. That same branch

[3]The format (a, b, c) is used here to present the data where a is a value derived from comparing gshare table sizes 1k and 2k, b is derived from table sizes 2k and 4k, and c is derived from table sizes 4k and 8k.



(a) Gshare PC Distribution (b) Gshare Miss Distribution

Figure 3.3: Distribution of occurrences and misses of fetched PCs in gshare's global history register.

"leads" the predictor to the right entry when combined with that same branch's PC. The second allows for two branches with behavioral patterns that may be too long or are lost due to large branch working sets to be distinguished due to additional local (PC) bits being used for indexing.

Next, we explore the effect of the history register width. Gshare was run on all benchmarks with a slight modification: we kept a pairing of the PC that contributed to the history register at each speculative update and the bit in the history register, Figure 3.3a. When examined, we find that roughly 30% of predictions are made when the current PC being predicted is not present in the global history register. Further, we also find that most misses occur at the extremes, having either 0 instances of a given PC in the history register, or having the register filled with the given PC (Figure 3.3b). Looking at Figure 3.3a, two anomalies are apparent. The first and easiest to explain is that for each history length, there seems to be roughly 10% of accesses having every bit of the register derived from the PC being predicted. This is caused by very tight, single backwards branch always taken loop and temporal locality. The second anomaly is the 12% of accesses at the 5^{th} column for the 1k and 2k (10 and 11 bit) gshare, and the mirror 6^{th} column for the 4k and 8k (12 and 13 bit) gshare. These spikes occur due to a two branch tight loop and are due to the high number of accesses to this loop, thus when exploring these two branches it is

likely that we find that one of these branches must be an always taken backwards branch, causing the loop to iterate enough times as to make its characteristics stand out, and the other can have any characteristic as long as both the fall through and target remain within the strongly taken loop.

```
... /* susan_corners() segment */
for (i=5; i<y_size-5; ++i)
  for (j=5; j<x_size-5; ++j) {
    ...
    if (n<max_no) {
      ...
      if (n<max_no) {
        ...
  } } }
```

Figure 3.4: Code from susan.c

When exploring the above hypothesis, regarding the two branches we found several more cases other than the strongly backwards taken loop with an internal branch. These cases are rare, and un-important to overall performance. Each time two tight loops following each other are executed more than 4-5 times, we found one instance of the history register containing 1/2 its bits from one PC, and the other 1/2 from another. The remaining cases were as predicted, and can be seen as an example from Susan's *Corners()* function (Figure 3.4). In this function, the branch returning us to the top of the loop for the second *for* loop is used frequently with the first *if* statement in the loop, which is often found to be false, which then leads to the history register to contain these two branches. Unfortunately, this case, like many others, is completely data driven. The value of n is dependent on the values passed into the function and the program by the user, thus interrupting the easy to predict NTNTNT... pattern.

Further, we explore how much history should be maintained. While it is believed that longer history does yield better prediction, no study has been done for gshare to decipher which, PC or history, bits contribute more to the improved prediction rate when doubling the table size. Each of the (1k, 2k, 4k, and 8k) gshare configurations was run with all possible history register widths and miss rate logged for each in Figure 3.5. The 0 entry is the one

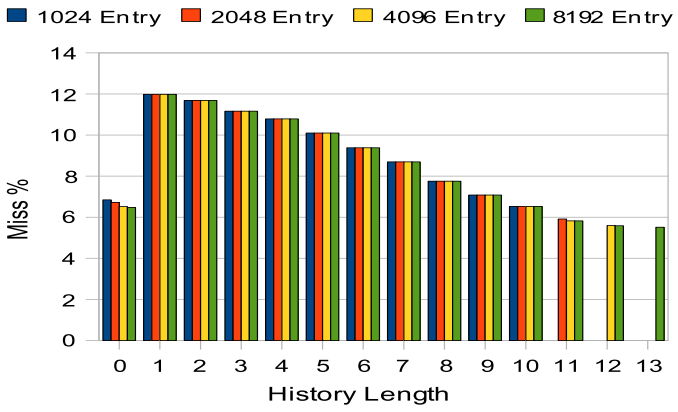


Figure 3.5: Average miss rate of gshare predictor with variable history register width.

corresponding to no history (bimodal indexing scheme). We found that when using less than 10 bits of history with the given benchmarks, we would have done better with the simple bimodal predictor. For every bit added after the first, we managed to get better prediction. As can be expected, the improvement per bit became less and less significant as more bits were already present in the global history register. The transition from 12 to 13 bits in the 8k gshare only reduced the miss rate by 0.08%, and it can be expected that another bit in a larger table would have near negligible miss rate reduction on the studied benchmarks.

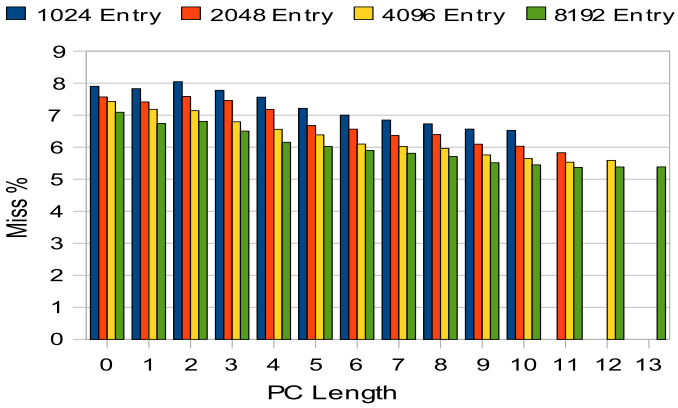


Figure 3.6: Average miss rate of gshare predictor with variable PC bits used for indexing.

Finally, we look at the relevance of the PC bits used in gshare. For this study we chose the 8k entry gshare with 13 bits of global history. Each run would increase the number of PC bits used to index from 0 (GAg) to the full 13 bits (Figure 3.6). Unlike the variable history length, increasing the number of PC bits used along side with the maximum history bits provides positive results in almost every case. Two small exceptions exist at 2 bits and again at 12 bits. Both cases are caused by a few benchmarks with strong global correlation that was lost when the additional PC bits were used.

3.3 Local Branch Predictor

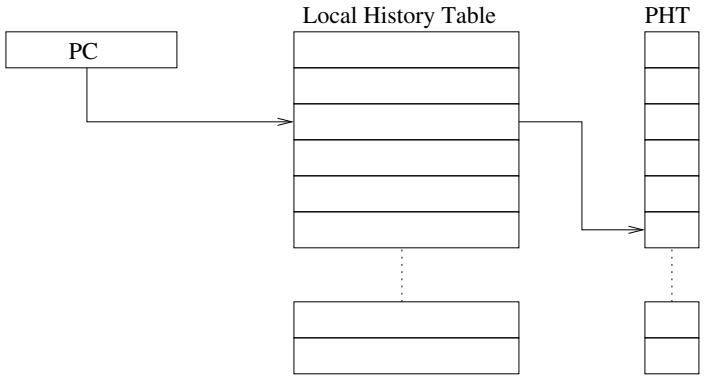


Figure 3.7: The 2 Level predictor configuration

The local predictor, or 2 Level predictor, is really a variation of the gshare predictor in which several local histories are kept instead of the single global history register. We have previously found that gshare derives better prediction when more instances of the current PC's history is present in the global history register. The local predictor, Figure 3.7, is designed to leverage off just that observation. The design of the local predictor for the purpose of this paper is a slight variation of the design given by Yeh and Patt in [3]. First, the current PC is used to index into a table of local history registers. Next, like the GAg predictor, the history register is used to index into the Pattern History Table (PHT). It is this two step design that earned the local predictor its second name "2 level predictor." Interestingly, while gshare out-performed the GAg predictor by using the xor of the PC with the global history register, the 2 Level predictor uses the local history to directly index into

the PHT. This works well since the local history was indexed via the PC of the branch, thus already generating a correlation between the PC and the history register used to index into the PHT. Secondly, 2 separate branches with similar history patterns will then be mapped into the same PHT entry, thus conserving space in the PHT. Due to its 2 level design combined with time constraints, the local predictor cannot have the same PHT size as other predictors which forced us to consider a wider range of permutations for table sizes seen in Table 3.5.

Table 3.5: Local Predictor Miss Rates

Lvl 1 Size	Lvl 2 Size	Avg. Miss	Std. Dev.	Lvl 1 Size	Lvl 2 Size	Avg. Miss	Std. Dev.
512	512	5.29%	4.93	1k	512	5.11%	4.94
512	1k	5.16%	4.95	1k	1k	5.02%	4.96
512	2k	5.11%	4.96	1k	2k	4.97%	4.97
512	4k	5.04%	4.97	1k	4k	4.90%	4.99
512	8k	5.01%	4.99	1k	8k	4.87%	5.01
2k	512	5.06%	4.94	4k	512	5.03%	4.94
2k	1k	4.95%	4.96	4k	1k	4.92%	4.97
2k	2k	4.91%	4.97	4k	2k	4.88%	4.98
2k	4k	4.84%	4.99	4k	4k	4.81%	4.99
2k	8k	4.81%	5.01	4k	8k	4.79%	5.01
8k	512	5.02%	4.95				
8k	1k	4.91%	4.97				
8k	2k	4.88%	4.98				
8k	4k	4.81%	4.99				
8k	8k	4.79%	5.01				

Comparing each of the local predictors to the next we find two trends similar to those we have found in every other predictor. A larger predictor yields slightly better prediction as well as a lower standard deviation for the given benchmarks. Similar to the GAg and gshare predictors, the local predictor improves faster as the history used is lengthened instead of the number of PC bits used to index. In fact, we can see almost no improvement in miss rate as the first level of the local predictor is increased beyond 4k entries (0.01% for the 512 and 1k second level sizes). Alternatively, an increase from 4k to 8k in the second level, consistently yields a 0.02-0.03% improvement for all first level table sizes.

3.4 Tournament Predictor

Since all the predictors described above can be configured to consume roughly the same space on chip and run within the same clock cycle, McFarling[2] suggested that we combine the advantages of several predictors. The tournament predictor as described in McFarling's paper was originally designed to use one of each category of branch predictors: a local and global. While his choice was to use the local and gshare predictors, this section will focus on a tournament predictor that uses the bimodal and gshare predictors. Using the bimodal predictor simplifies the table access patterns and the amount of data that must be recorded to create a good analysis of the tournament predictor. At the end of the section we will briefly cover a few configurations of the tournament predictor utilizing the local branch predictor to compare the difference relative to the bimodal predictor.

3.4.1 Bimodal/Gshare Tournament Predictor

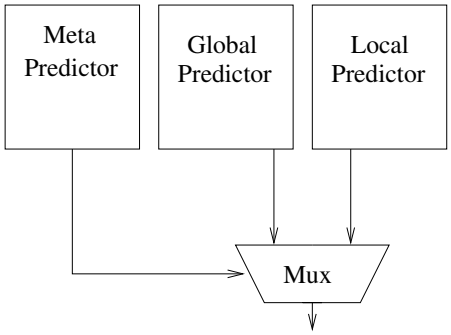


Figure 3.8: Tournament predictor configuration

Figure 3.8 shows the configuration of the tournament predictor. The predictor is simply a combination of any two of the predictors mentioned above, as well as a Meta predictor to make the selection between the two outcomes. The Meta predictor itself is nothing more than a bimodal predictor with a new meaning assigned to the 2 bit outcome, 00 and 01 are used to signal the use of one predictor's result to be used while 10 and 11 are used for the the other predictor. Next, the meta predictor's update rules must be modified from the standard bimodal predictor. Updates only take place when the two predictors provide different results.

In which case the state of the meta predictor will shift in favor of the predictor that made the correct prediction. The driving force behind this larger while relatively simple predictor is the availability of power and area for a slight gain in performance.

Several configurations have been made for the tournament predictor with varying bimodal, gshare, and meta predictors (Table 3.4.1). Included in Table 3.4.1, are the average miss rates excluding the adpcm benchmarks, as well as the average miss rate of the gshare predictor excluding the adpcm benchmark. When running the tournament predictor, we find that the meta predictor is extremely sensitive to branches in the adpcm benchmark. We have previously seen that both the *coder* and *decoder* executions of the adpcm benchmark yield very poor branch prediction results in the bimodal predictor and to some extent the gshare predictor. The net result is one that provides slightly better performance in the adpcm benchmark when compared to the bimodal predictor, but considerably worse than the gshare predictor. These results, when averaged with the remaining benchmarks, skew the average to one which is, in some cases, greater than that of the gshare predictor of similar clock cycle timing (such as the 4k/4k/4k configuration yielding a 5.88% miss rate compared to the simple 4k gshare with 5.77% miss rate). Due to outliers such as this, it is necessary to present results in two distinct categories: *Normal* and *Deviant*. In this particular case, the tournament predictor encounters "odd" behavior in the adpcm benchmark, having an average miss rate several standard deviations beyond the average. This behavior does not reflect on the performance of the predictor as a whole, but should not be ignored. All the benchmarks other than adpcm are then placed in the *Normal* category, while adpcm will be considered *Deviant*, having a miss rate greater than one standard deviation away from the average of all the benchmarks. To allow for a fair comparison of branch predictors, previous base predictors were re-organized into the *Normal* and *Deviant* classifications and miss rates presented in Appendix A. Further, the bizarre behaviour of the adpcm benchmarks was further studied and is presented in Appendix D.

When the two averages, excluding the adpcm benchmark, presented in Table 3.4.1 are reviewed it can be seen that an another anomaly occurs in the 1k/1k/8k and 2k/2k/8k configurations. In both cases the critical path is the access to the 8k meta predictor and thus allows for a comparison with the 8k gshare; and in both cases, the tournament predictor was outperformed by the simpler 8k gshare. Such behavior reveals that the meta predictor, while obviously aiding in the reduction of miss rate, is not the critical element of the tournament

Table 3.6: Tournament predictor miss rates in *Normal* and *Deviant* categorizations.

Bimodal Table Size	Gshare Table Size	Meta Table Size	Avg. Miss Rate	Normal Avg. Miss Rate	Deviant Avg. Miss Rate
1k	1k	1k	6.30%	4.72%	24.41%
1k	1k	2k	6.28%	4.70%	24.41%
1k	1k	4k	6.23%	4.65%	24.41%
1k	1k	8k	6.22%	4.64%	24.41%
2k	2k	1k	6.07%	4.52%	23.85%
2k	2k	2k	6.07%	4.52%	23.85%
2k	2k	4k	6.03%	4.48%	23.85%
2k	2k	8k	6.03%	4.48%	23.85%
4k	4k	1k	5.88%	4.33%	23.65%
4k	4k	2k	5.88%	4.33%	23.65%
4k	4k	4k	5.88%	4.33%	23.65%
4k	4k	8k	5.87%	4.32%	23.65%
8k	8k	1k	5.73%	4.19%	23.35%
8k	8k	2k	5.73%	4.20%	23.35%
8k	8k	4k	5.73%	4.20%	23.35%
8k	8k	8k	5.74%	4.21%	23.35%

predictor. Further, it can be seen that in the case of the tournament predictor average miss rate excluding the adpcm benchmark, the transition from the 1k/1k/1k configuration to the 1k/1k/4k yields an improvement of 0.07%. While the 1k/1k/1k to 2k/2k/1k move, having the same data size increase, yields an improvement of 0.20%.

The poor performance of branch predictors caused by data driven branches found in adpcm is found to dominate the miss rate of the tournament predictor just as it did with the bimodal, gshare, and local predictors. While all other benchmarks achieve a lower miss rate when compared to the gshare predictor, adpcm does not. In fact, the tournament predictor provides only a slight improvement over the bimodal predictor. We have already recognized that the poorly predicting branches of the adpcm benchmark are poorly predicting since their outcomes are essentially random. We then find that with both of our "simple" predictors (bimodal and gshare) we will also have a random prediction pattern, while making some improved predictions from other frequently executed and well behaved branches. When run in parallel, we find that for some short sequences the bimodal predictor will out-perform

the gshare predictor. Other sequences are better predicted by the gshare predictor. We will now refer to these sequences as *phases*[4]. By referencing previous data from adpcm, we find that for these pseudo random branches, the gshare predictor will out-perform the bimodal predictor over the length of the execution. The ideal scenario would be to use the right predictor for the right phase of execution. Unfortunately, these phases can be short and cannot be detected ahead of time. While the ultimate goal of the tournament predictor is to detect such phase changes and adapt dynamically, the short burst phase changes throw the predictor off and cause additional penalty when compared to using gshare only. In the case of the 1k/1k/1k configuration we find several branches, all having "random" prediction paths, that incur many phase changes within the meta predictor (Table 3.7). It is also important to note that none of these phase changes were caused by contention (While there were other branches which indexed into the meta table entries in question, none were within temporal locality of the branches causing the poor performance). Ideally, the number of phase switches should be kept to a minimum, with the optimal being 0 or 1 phase changes.

Table 3.7: Number of phase switches in the Meta predictor of the 1k/1k/1k tournament predictor configuration from adpcm.

Entry	Phase Switches
305	76996
309	25650
315	75923
321	81038
345	67279

In a very similar fashion to the evolution of the bimodal predictor from the "predict same as last" predictor, it would make sense to modify the meta predictor so that short phase changes do not alter its state and cause additional mispredictions. Only long phases which favor the bimodal predictor will then shift the meta predictor out of the gshare's favor. This can be done by increasing the number of updates to the meta predictor required to detect a phase change. Thus, short phase changes that would have previously moved the prediction from one predictor to the other would no longer cause the swap. To achieve such

[4]A detailed example of phase changes in the adpcm benchmark can be found in Appendix D

a behavior we must simply increase the number of bits used by the bimodal meta predictor. In this test (Table 3.4.1), the meta predictor was inflated to use 4 bits and thus 16 states (8 representing the bimodal predictor and 8 representing the gshare predictor). The transition between states is the same as the previous 2 bit machine, but simply lengthened to include more confidence levels beyond *Strongly* and *Weakly*. Since the tournament predictor failed to out-perform the gshare predictor in the larger configurations, 8k table sizes became the focus of this study. It is important to note that if area is to be conserved, the 4k meta predictor with 4 bits per entry is of equivalent size to the old 8k meta predictor with 2 bits.

Table 3.8: Tournament Predictor Miss Rates with 4 bit Meta table entries.

Bimodal Table Size	Gshare Table Size	Meta Table Size	Avg. Miss Rate	Normal Avg. Miss Rate	Deviant Avg. Miss Rate
1k	1k	512	6.19%	4.64%	24.12%
2k	2k	1k	5.99%	4.44%	23.72%
4k	4k	2k	5.78%	4.24%	23.45%
8k	8k	4k	5.64%	4.12%	23.14%
8k	8k	8k	5.64%	4.12%	23.14%

As can be seen in Table 3.4.1, not only did increasing the meta predictor's number of states help decrease the miss rate of adpcm to near that of the gshare results but it also decreased the miss rate of all the other benchmarks. It should also be noted that just like the 2 bit meta predictor, the size of the meta predictor hardly affects the performance of the tournament predictor when using larger predictors. This allows us to compare the 8k/8k/4k (4 bit) configuration with the 8k/8k/8k (2 bit) configuration, which have the same size.

3.4.2 Local/Gshare Tournament Predictor

The tournament predictor was reconfigured to use the local branch predictor due to its ability to predict a category of branches which exhibit poor performance with both the bimodal and gshare predictors. Four configurations of the new tournament predictor were run to completion to provide the following results in Table 3.9. For simple execution, when the local predictor size is specified, it is divided evenly between the two levels of tables.

Looking at the results in Table 3.9 reveals that while the use of the local predictor instead of the bimodal predictor does improve the prediction rate of the tournament predictor, even

Table 3.9: Local/Gshare Tournament predictor miss rates with 2 bit meta table saturating counters.

Bimodal Table Size	Gshare Table Size	Meta Table Size	Avg. Miss Rate	Normal Avg. Miss Rate	Deviant Avg. Miss Rate
1k	1k	1k	5.28%	4.03%	19.64%
2k	2k	2k	5.01%	3.74%	19.59%
4k	4k	4k	4.88%	3.61%	19.54%
8k	8k	8k	4.79%	3.51%	19.54%

when compared to the 4-bit meta tournament predictor, it performs only slightly better than simply using the local predictor alone. In fact, in each of the 4 cases, the miss rate is exactly 0.01% lower than its local predictor equivalent. Making the assumption that short phases are correlated to the higher than expected miss rate, we decided to take the same approach we took with the bimodal/gshare tournament predictor. We modified the meta predictor again to use 4 bit counters (Table 3.10) and used table sizes for the meta predictor of equivalent size (1/2 the number of entries). The results, like the original tournament predictor, are considerably better.

Table 3.10: Local/Gshare Tournament predictor miss rates with 4 bit meta table saturating counters.

Bimodal Table Size	Gshare Table Size	Meta Table Size	Avg. Miss Rate	Normal Avg. Miss Rate	Deviant Avg. Miss Rate
1k	1k	512	5.15%	3.94%	19.09%
2k	2k	1k	4.89%	3.66%	19.10%
4k	4k	2k	4.75%	3.51%	19.09%
8k	8k	4k	4.64%	3.39%	19.07%
8k	8k	8k	4.63%	3.38%	19.07%

CHAPTER 4

MOVING TO A BETTER PREDICTOR

4.1 Bimodal to GAg

GAg, compared to a local predictor such as the bimodal predictor, takes advantage of the global pattern to determine the predicted pattern. When looking at specific branches and their miss rate, we find that a few branches derive great benefit from the global prediction. These branches generally exist in a tight loop and depend on the previous iteration of the loop, but not data. A simple example is one of a branch having a TNTN pattern existing within a tight loop iterated via an always taken backwards branch (TTTT pattern). In this case, the bimodal predictor would get almost every instance of the backwards branch, and miss 50% of the time on the switching branch at best. In a worst case scenario, the bimodal predictor could miss 100% of the TNTN branch by constantly switching between the Weakly Taken and Weakly Not Taken states and thus predicting wrong every time. On the other hand, GAg's global history would hold a pattern of TTNT and will repeat. In this pattern, the first and third elements make the TN cycle of the branch in question, while the second and fourth elements make the TT pattern. This means that once the predictor "warms up" the miss rate will be 0%.

The example given above is very simple, using a simple pattern with a 2 branch working set. Unfortunately, simple global pattern loops such as these with such small working sets are either rare or do not iterate enough to skew the total branch prediction in the benchmarks studied. Other simple branches can greatly suffer from the global indexing scheme. For example: the same always taken backwards branch from the previous example, that would normally have a near 0% miss rate, could be thrown off by the history of an irregular pattern. If the second internal branch has a pattern that is completely data dependent, while using random data, then the same pattern in the history register does not necessarily determine

the future pattern. Instead, data loaded in that loop iteration will determine the pattern. Thus the history generated by the previous branches is itself random, and provide a random prediction for the next branch. It would then take a long time before all the permutations of the random branch can be present in the global history register and the PHT is trained. When using a 10 bit global history register and a 2 branch working set described above the random branch can occupy 5 bits and either take up the even or odd bits, thus for all the permutations to occur it would take at least 2^6 executions of this loop to train the PHT.

A second example, which is technically possible but was not discovered in the benchmarks studied, is the result of a pattern that is too long to be contained in the history register but will not necessarily cripple the result of the always taken branch. In this case the pattern simply takes a long time to reach the "warmed up" state. Finally, there can be a problem with contention when the pattern is too long. If the pattern for a branch is along the lines of TNTTTNTTNT, we can see the first sub pattern TNTT repeats a second time to be followed by NT, and that branch is contained within an always taken backwards branch. Finally, if this example was to use an 8 bit history register, the global history register would be the same at the end of the first and second instance of the sub pattern (TTNTTTTT). The first time, this pattern would lead to to the same pattern being generated again. The second time would lead into the NT sub-pattern and mispredict. Further, this misprediction would cause the second sub-pattern to miss again due to the entry corresponding with TTNTTTTT to be updated to the Weakly Not Taken state.

4.2 GAg to Gshare

A key aspect of gshare is the even distribution of the PC's used in each table entry. Unlike previous predictors discussed, gshare is very nearly evenly distributed. When using a 1k, 2k, 4k and 8k prediction tables we find an average percentage of PCs indexed per entry to be 0.098%, 0.049%, 0.024%, and 0.012% with standard deviations of 0.042, 0.028, 0.019, and 0.013 respectively. These values show that in each case where the table size doubles, the percentage of PCs indexing into each entry is halved. When comparing these values to the bimodal or GAg predictor, PHT distribution we can see that the gshare predictor, having a standard deviation roughly half the mean, has a much more even use distribution of the PHT. The fact that the standard deviation differences do not keep to the same exact pattern reflect

on the increase in number of entries having 0 accesses with larger tables. It then follows that the improvement made by the gshare indexing scheme vs. the GAg indexing scheme is derived from the OXR of the PC into the global history register thus allowing additional locality to the PHT indexing previously gained from a longer global history register.

4.3 Gshare to Local

While the move from the single global history register to a table of local history registers seems intuitive, it yields remarkable results. As previously mentioned, the gshare predictor derives most of its power from a longer history register. In most cases, due to the presence of the same branch in that register. When several history registers are then used, with contention only present due to PC indexing conflicts, the local predictor behaves exactly as expected, providing better average results for each benchmark. Rare cases of individual branches were found to have decreased performance when making the change from the gshare to the local predictor, each with a degree of correlation with the global path. A few examples of such branches are provided in Table 4.1. Overall, it was found that 4.5% of static branches suffer from the switch to the local branch predictor for a total of of 1.8M additional mispredictions.

Table 4.1: Some examples of branches which perform worse with the local predictor vs. the gshare predictor.

Benchmark	PC	8k gshare num. Misses	4k/4k local num. Misses
mpeg2 encode	120006f8c	103,912	189,802
g721 encode	120002c90	15,291	86,254
g721 decode	120001558	110	45,016
gsm toast	120006d9c	4,807	61,084
patricia	120059760	98,004	132,339
adpcm compress	1200004d4	157,188	187,901
FFT	12004c06c	1,713	22,960
lame	1200524bc	583,507	601,345

In the drastic case of g721 decode, at PC 120001558, we find a total degradation in miss rate of nearly 41,000% from 110 misses to 45,016. Further examination of the code reveals

a very strong correlation with a previous *if...else if...else* statement. Where the gshare predictor was able to use such correlation, the local predictor was only able to assume that this branch outcome is correlated to its previous outcomes and failed to achieve the high hit rate of the gshare predictor. While the g721 decode benchmark did suffer from a few lost correlations, it is important to recognize that the local predictor did perform better as a whole for the benchmark, bringing the miss rate down from a 5.22% to a 3.06%.

Next, a study was conducted, similar to the one done for the gshare predictor in Section 3.2, examining the top 10 missed branches of one configuration and comparing it to another. For the purpose of a simple comparison, the 1k gshare and 512/512 local predictors were chosen to represent the changes in the top 10 missed branches. While previously we have seen differences such as 1.6 new branches and 4.1 shifted branches, the gshare to local predictor comparison reveals that the new indexing scheme causes an average of 3.0 new branches to be placed in the top 10 list, while roughly maintaining the existing 4.1 average shifted branches (4.5 for the gshare to local comparison). Having nearly doubled the number of branches which are new to the top 10 list we find further support that the local predictor does not simply provide better prediction across all branches, but is simply better at predicting different types of branches.

4.4 Components to Tournament

As mentioned before, the tournament predictor is a relatively simple combination of the bimodal or local predictor as well as the gshare predictor. There are several scenarios in which the tournament predictor benefits from, relative to its individual components. The first, is applicable when using the bimodal/gshare combination. Having the 2-bit saturating counters with a simple PC based indexing scheme of the bimodal predictor allows bypassing of the *warm-up* time required before the gshare predictor is capable of delivering accurate results. Since gshare requires several bits of history for indexing into the appropriate table entry, as well as having several table entries contributing to a branch's prediction due to varying history register values, it takes several iterations of the branch PC before relevant predictions appear in the gshare predictor. On the other hand, for simple branches, the bimodal predictor requires 1 or 2 iterations of the branch as its *warm-up* phase. While the gshare predictor is collecting data to generate its generally superior predictions, the bimodal

predictor is more likely to produce relevant predictions. If this is the case, the meta predictor will recognize the bimodal predictor's correct predictions and update to reflect that gshare should not be used. Eventually, the gshare predictor will begin producing more correct results than the bimodal predictor and, again, the meta predictor will be updated to reflect the new configuration.

The second, and less likely scenario, is that the bimodal predictor simply out performs the gshare predictor on average for the entire or a long portion of the branch's execution. In which case, the tournament predictor has the advantage of using the bimodal predictor instead of the gshare. Several instances of these branches have been found, but in no case was the bimodal miss rate more than 1% lower than the gshare miss rate for the average execution of significant branches[1]. On the other hand, this second case is a very viable scenario for the local/gshare combination tournament predictor. As we mentioned before, the local predictor is very good at capturing branch behavior that is missed by the other base predictors covered thus far. Mentioned in the previous section, we saw that there were roughly 4.5% of all static branches which perform better with the gshare predictor when compared to the local predictor, for a total of 1.8M dynamic mispredictions. More statistics reveal that these 1.8M additional mispredictions are collected during 73M dynamic executions of the above mentioned 4.5% of static branches (2.5% potential improvement). Alternatively, the switch from bimodal to gshare we find that 15.0% of static branches perform better with the bimodal predictor for a total of 1.6M dynamic mispredictions out of 120M dynamic executions (1.3% potential improvement). Finally, comparing the bimodal to the local predictor reveals that while 6.3% of static branches do perform better with the bimodal predictor, they only account for 0.76M dynamic mispredictions out of 91M dynamic executions (0.84% potential improvement). Thus by classifying branches as *improved*, *constant*, and *degraded*; and collecting statistics on each class, specifically *improved* and *degraded* branches, we find that if two predictors are to be used along with a meta predictor, the best performing one should be expected to be the one with the largest contrast, i.e. the highest misses to executions ratio. Finally, the previous assumption is confirmed when examining the data presented in Chapter 3 and comparing the three ratios provided above, specifically the better

[1]Significant branches are branches which account for at least 5% of the total branch execution. This means that generally, branches which execute only a few times, will not be considered significant since even a 100% miss rate will not cause an increase in the overall miss rate.

prediction rate of the local/gshare predictor vs. the bimodal/gshare predictor. We also find that as long as the meta predictor avoids heavy contention, we are able to make use of this characteristic and enjoy the benefits of the local/gshare tournament predictor for these otherwise hard to predictor branches.

It was also found that a critical aspect in achieving better performance with larger prediction components it was necessary to optimize the meta predictor. Essentially ridding the meta predictor of short and harmful phase shifts, the new meta predictor's entry size was doubled to result in a 4 bit (16 state) predictor. The original, 8k/8k/8k (2 bit) configuration utilizing the bimodal/gshare predictors resulted in a 5.74% miss rate, compared to the 8k gshare with a miss rate of 5.66%. Likewise, the original 8k/8k/8k (2 bit) configuration of the local/gshare predictors resulted in a 4.79% miss rate, compared to the 8k (4k/4k) local predictor with a miss rate of 4.81%. The new configuration which eliminated most of the harmful phase changes in the meta predictor achieved a 5.64% miss rate in the bimodal/gshare configuration and a miss rate of 4.64% for the alternative local/gshare combination. While the 0.02% improvement upon the gshare predictor may seem insignificant, it is important to note that such a change confirms earlier statements referring to the tournament predictor as a trade-off between power/space and slight performance improvement. On the other hand, the local/gshare configuration yields a 0.17% improvement over the local predictor alone; which in high power, wide-issue, aggressive processors with a potential miss penalty of 10+ cycles could potentially be energy/cost effective. The difference between the miss rates of the two configurations of the local predictor is proof that analysis such as the one made in the previous paragraph regarding the behavior of specific branches when changing branch predictors can determine the best configuration of the tournament predictor.

Further, it was said that due to the fine grain improvements made by these more advanced predictors data should also be presented in the categories of *Normal* and *Deviant*. In this particular case, the adpcm benchmarks (both the Compress and Decompress) were considered *Deviant* due to their inability to out-perform gshare in the bimodal/gshare tournament predictor. When using these categories, for the configurations mentioned above, we find that the old 8k/8k/8k (2 bit) configuration of the bimodal/gshare resulted in a 4.21% miss rate, while the 8k gshare achieved a 4.44% miss rate. Already we find that for all benchmarks within the *Normal* category, the tournament predictor performs considerably

better. The new 8k/8k/4k (4 bit) configuration yields a miss rate of 4.12%, bringing the net drop in miss rate to 0.32% when compared to gshare (much higher than the overall average which yielded a 0.02% drop). The local/gshare configuration of the tournament predictor is also presented in the two separate categories, although in this case, the tournament predictor did outperform the local predictor alone.

CHAPTER 5

RELATED WORK

In the following sections we will cover other work that have been done towards improving the quality of branch prediction beyond the spectrum of the research done here. Each of these studies will be tied to trends revealed in this paper and placed in one of two categories. The first category includes the *Capacity Based* predictors, which are those predictors that rely on the initial trend studied in this paper revealing that larger predictors improve prediction accuracy. We will also discuss the limitations in this spectrum of research. The second category includes the *Access Pattern Based* predictors, which are predictors that use an indexing mode different from one studied in this paper.

5.1 Capacity Based Development

As seen before, the increase in capacity will yield a better prediction rate. While different predictors will benefit at differing rates from a similar increase in predictor size, it is clear that increasing the predictor capacity will be the simplest way to decrease the miss rate. Unfortunately, there is a limitation to the gains made by capacity increase. Capacity increases aid branch prediction in two distinct ways: decreased contention and ability to retain longer history. We have shown that as capacity increases the benefits of lowered miss rates are diminished. This phenomena is caused by a threshold set by the nature of branches and the branch predictor design. When contention is eliminated and the history is long enough to retain the longest patterns we will still suffer from random branches and warm-up phases. Finally, it is also important to realize that while we may be able to fit a large branch predictor on chip, the additional size may limit the rate at which we are able to make predictions and make it impossible to achieve a prediction in one cycle.

5.1.1 Caching Branch Predictor

The caching branch predictor[5] is an alteration made on the gshare predictor. Based on the observation that gshare benefits greatly from longer history, and thus a larger PHT, the caching predictor is designed to cache the most recent entries used by the predictor and attempt an access from the faster Pattern History Table Cache (PHTC) before accessing the larger PHT. Like any cache, there is a chance that the searched item is not found. For such an instance, Jiménez et. al. included a small Auxiliary Branch Predictor (ABP) which is accessed in parallel to the PHTC. In case of a PHTC miss, the ABP prediction is used. On an update, all the structures require an update.

In the case of gshare, no performance gain is made by the caching predictor. While there is no decrease in miss rate, there has been no study in altering the specific branch prediction scheme used to index the PHTC or the ABP. Also, it is not mentioned in [5] that this predictor scales well for faster, more aggressive fetch engines due to the caching scheme that makes use of temporal locality of branches (i.e. tight loops).

5.1.2 Cascading Look-ahead Branch Predictor

The cascading look-ahead branch predictor[5] is a modification of the look-ahead branch predictor[21]. The look-ahead configuration is based on average code having a branch instruction every 3-4 non-branch instructions. This statistic allows a prediction to begin early and consume several clock cycles before being needed. To achieve this behavior, a simple alteration to the gshare predictor is made. First, the global history register is updated speculatively with the prediction made by the predictor as soon as it is made. Second, the PC used to index into the PHT along with the history register is no longer the PC of the current instruction fetched but is replaced with the PC of the target instruction from the BTB. This of course forms a dependency between the BTB hit rate and the correct prediction of the cascading look-ahead predictor. Finally, to allow for back to back branches, two separate PHTs are used. The first PHT (PHT1) is a smaller table capable of a single cycle access, while the second PHT (PHT2) is a larger table requiring 2-3 cycle access. After a prediction is made, both PHT1 and PHT2 are accessed using the speculatively updated global history register and the target PC from the BTB. When the next branch instruction arrives then the prediction from PHT2 is used, assuming access has completed. If the branches were not

spaced far enough apart, the prediction from PHT1 is used instead.

While this scheme out-performs the caching predictor in the data presented in [5], it does not account for wide-issue instruction fetch, which can cause one or more branch instruction to be fetched each cycle. Such a fetch engine can cripple all the gains made by the cascading look-ahead predictor reducing it to a simple 1 cycle gshare. The downfall of the cascading look-ahead branch predictor is its reliance on spacial locality, which becomes less relevant with more aggressive processor designs. While, on the other hand, the caching branch predictor is based on temporal locality and should not suffer from the more aggressive processor.

5.1.3 Overriding Branch Predictors

The overriding branch predictor is similar in design to the cascading look-ahead predictor. Both predictors contain a fast access PHT1 and a slower more accurate PHT2. In the case of the overriding predictor, prediction begins on the branch instruction's PC. The fast and generally less accurate PHT1 prediction is used to fetch the next instruction while the PHT2 look up runs to completion. When the prediction arrives from the PHT2 a simple comparison is made between the two predictors allowing the PHT2 to override the less accurate PHT1 prediction in the case where the two disagree. The penalty for the override depends on the delay of the PHT2; an n cycle PHT2 would complete $n-1$ cycles late, and would generally require 1 cycle to squash the incorrectly-fetched instructions. As pipelines become deeper and branch resolution is distanced from the fetch stage such an override becomes more beneficial. For example, it is not unrealistic to assume a 9 cycle delay between the fetch of a branch and the branch resolution in a complex, deeply pipelined processor. In such a case, a 2 or even 3 cycle penalty for a rare override may far out-weigh the alternative 9 cycle miss.

The overriding predictor is well geared towards deeply pipelined and complex processors. In fact, the gains made by the override predictor are enhanced as the penalty for a miss becomes greater. While at first glance it may appear that the override predictor could benefit from a similar access scheme to the cascading look-ahead prediction scheme, we find that the above mentioned gains made by the overriding predictor are made in the same style processor as the one that would cripple the benefits made by the cascading look-ahead predictor. In other words, a wide issue deeply pipelined design that would benefit from the override predictor is likely to fetch a branch on each cycle which would mean that the PHT1

would have to be used every time and would not be likely to be bypassed.

5.1.4 Pipelined Branch Predictors

In his work, *Reconsidering Complex Branch Predictors*[4], Jiménez sought to improve branch prediction in a similar fashion as the caching predictor, with a reduced latency for large PHT tables. Instead of using a cache to retain recently used PHT entries, the PHT table is accessed on each cycle to fetch the entries which could possibly be called upon at the time a branch is fetched. If for example, the PHT access requires n cycles, then 2^n entries are fetched. Since the prediction is pipelined, wide issue architecture requires a modified version of the predictor. It is claimed that by using an old version of the history register, more entries could be fetched into the PHT buffer. This technique has been proven to have minimal degradation of the prediction accuracy.

While this technique presented appears to provide strong support for the pipelined predictor for wide issue machines (the IPC count when accounting for the cycle delay of the branch predictors studied is highest for their *gshare.fast*), the study leaves out the predictor most similar to itself, the caching predictor. Since both predictors extract a small segment of the PHT for a quick access, it would appear that they would both provide a similar miss rate within a single cycle. A further study should have been made on energy consumption. It appears that the similar behavior, when compared to the caching predictor, would make for an interesting comparison. It would be my personal speculation that the caching predictor would yield similar performance with considerably lower energy consumption.

5.2 Access Pattern Based Development

As mentioned in the previous section, increasing capacity of a branch prediction yields fast and easy results but at the cost of access time. When looking at the development of new styles of branch predictors it becomes apparent that the largest gains occur when using a new indexing scheme for the PHT. At times, a new indexing scheme will out-perform another while using 1/8 of the transistors when compared to the old scheme (ex: 1k gshare predictor yields a miss rate of 6.76%, while an 8k bimodal predictor yields a miss rate of 6.84%).

5.2.1 Bi-Mode Predictor

The bi-mode predictor[16] is a table access alteration based on the same principals of larger prediction tables. Currently, there are two driving reasons for increasing a prediction table size. The first is to lower interference between branches or patterns accessing the same PHT entry. The second is to allow for longer history and thus detect better correlation between previous branch executions and the current prediction. The bi-mode predictor is focused on the first. Similar to the tournament predictor, the bi-mode predictor uses a simple PC indexed 2-bit counter table to select between the predictions made by two other tables. Unlike the tournament predictor, both tables are indexed using the same PC and global history hash and differ only in the update policy. To reduce destructive aliasing between branches, the bi-mode predictor only updates the entry in the table that was selected for the prediction. Next, the meta predictor is always updated to maintain which of the PHT's is more able to provide a better prediction (similar to the tournament predictor). Finally, it was concluded that the bi-mode predictor using the gshare indexing scheme does out-perform the size equivalent gshare predictor.

5.2.2 (M,N) Correlation-Based Predictor

The (M,N) correlation-based predictor[13] uses an M bit global history register in a unique way. First, like the bimodal predictor, the branch PC is used to index into a table containing N entries. Unlike the bimodal predictor, each entry now contains 2^M 2-bit saturating counters. Using the global history register, the appropriate counter is selected and used for the prediction. Simply illustrated, the low order significant bits of the PC are used as the high order bits of the PHT index. This index is then concatenated with the most recent global history bits to form an index into a one dimensional PHT simulating the $Nx2^M$ PHT described earlier. In their research, the correlation-based predictor was compared to the simple bimodal predictor and was found to provide a better prediction rate. Later, in [2], it was shown that the gshare XOR scheme is a better indexing scheme when compared to the concatenation of PC and global history.

5.2.3 Data Based Correlation Predictor

The data based correlation predictor, or Branch Difference Predictor (BDP)[15], attempts to use previous data values to drive the prediction for hard to predict data dependent branches. For each branch comparing two registers, the difference between the register values is stored in a Branch Difference Table (BDT) and indexed by the branch PC. Another table, the Branch Count Table (BCT) is then used to store the number of in flight branches of each branch PC stored in the BDT. Combined, these two tables, the BDT and BCT, are called the Value History Table (VHT). The BCT essentially provides a method of telling how reliable the data stored in the BDT is (the higher the value in the BCT, the less reliable the data in the BDT). A third structure, the Rare Event Predictor (REP), is then used to make the prediction for branches which are detected to have mispredicted with the backing-predictor and hit in the VHT. In the benchmarks studied (compress, gcc, go, ijpeg, and li) the BDP is reported to have roughly 10% reduction in mispredictions when compared to gshare or the Bi-Mode predictors.

5.2.4 Address Based Correlation Predictor

The address based correlation predictor[14] goes a step further than the data based correlation predictor and defines a correlation between data address and the data itself, thus a correlation between the address of the data and the prediction. In some long latency loads ($>$ 100 cycles), Gao et. al. found that the address of the data can be often used to predict the result of the branch. The benchmarks selected were ones which display a heavy dependency on long latency loads, thus allowing for the greatest potential gains. One of the benchmarks, *parser*, was found to perform poorly due to varying data structures. As a result, data for that benchmark was not collected until the first 700M instructions were executed. On average, total energy savings of 5.2% were reported due to a 4.2% reduction in total cycle count. While these savings seem impressive, it is important to note that they only represent data collection from benchmarks with emphasis on high latency loads and in the case of *parser*, only parts of the benchmarks that benefit from the predictor. When collecting data for *equake*, it was found that total energy consumption was increased by 0.4%. Combined, these facts reveal that while this predictor design works well with the given set of branches, the inability to turn the predictor *on* or *off* will cause the overall power demands

of the predictor to outweigh the benefits made to 2-3 specific branches per benchmark.

5.2.5 Agree Branch Predictors

The agree branch predictor[17] is an alteration of the meaning of the state machine described by the 2 bit saturating counters of the PHT in the gshare predictor. Each entry in the BTB is appended a single bit, biasing bit, which is set according to the first result of the branch resolution after the branch has been placed in the BTB. In other words, if the branch was not in the BTB and was found to be taken, then the branch would be placed in the BTB. The second time the branch is executed, the biasing bit will be set in the BTB with the calculated direction of the branch. Next, we treat the 10 and 11 states of the counter as the *agree* states and the 01 and 00 states as the *disagree* states. This means that two branches, indexing into the same counter, would hopefully both saturate the counter towards the agree state; even if their biasing bit is different.

The approach taken above completely depend on the initial value of the biasing bit. Due to the high affinity for dynamic branch prediction, the agree predictor was implemented as mentioned above (using the second execution of the branch to set the biasing bit). Alternatively, a static approach could potentially be very helpful in this case. An older study[12] was done with static branch prediction. In their study, each branch instruction was appended with a single bit to signify the most common direction of the branch during an execution trace done at compile time. Such training provides poor results when compared to the dynamic approach, as well as require slight changes to the instruction set. Regardless, this approach was viable for very small embedded application that did not have the hardware budget for a dynamic branch predictor. Today, embedded processors are relatively large and might be a valid field of application for a combination static/dynamic approach such as using the execution trace to generate the biasing bits which are then embedded into the instructions and used by the agree predictor.

5.2.6 Neural Branch Predictors

Neural predictors, such as the perceptron predictor described by Jiménez and Lin [18][19], attack a different part of the branch predictor, the saturating counter. Ultimately, we have found that in all the prediction schemes the two restrictions to better prediction are conflicts and access to longer history. While a saturating counter PHT allows for at most $\log_2 N$ bits

of history, where N is the number of entries in the PHT, the neural predictors are bound by a linear growth function. Each neuron is indexed by a PC based hash function, and is then allowed longer history since the output of the neuron is an arithmetic function based on the history provided. In the study done by Jiménez and Lin, it was found that the perceptron neuron was best suited for branch prediction due to quick training and relatively simple design, hence the name "perceptron predictor". That function, $w_0 + \sum_{i=1}^{n} x_i w_i \Rightarrow prediction$, is calculated by each neuron given a set $History = \{1, x_1, ..., x_n\}$ where x_i can be either 1 or -1. The elements w_i are 8 bit registers retained within each neuron as a set of $Weights = \{w_0, w_1, ..., w_n\}$. It is important to notice that the element x_0 is always 1 and therefore allows for a bias weight that is not based on global history to be used in w_0. Reported in their research, the neural predictor provides a 26% improvement over gshare which would equate to roughly 4.2% miss rate (including the adpcm benchmarks).

CHAPTER 6

CONCLUSION

While larger branch predictors and longer recorded history are proven to provide more accurate predictions we find that, with current processor design trends, it becomes infeasible to access such large tables within a single cycle. Other predictors, such as neural predictors, allow the use of longer history with small tables, but at the cost of circuit complexity and potentially slower clock cycles. Concurrently, modern applications demand a low power processor while maintaining high performance. Early research[10] focused on reducing the potential cost of a branch misprediction by utilizing delay slots (instruction to be executed after a branch regardless of the true branch outcome). These delay slots, while useful for simpler processors, become nearly impossible to fill for a wide-issue aggressive design. McFarling and Hennessy report that a single delay slot can be filled 70% of the time, while a second delay slot can only be filled 25% of the time, making delay slots a dead end for wide-issue processors and an already refined area of research for embedded processors.

Current trends lead us to explore the possibility of reducing the prediction miss rate, thus minimizing the probability of the potential cost of the branch misprediction. Some research has gone into being able to access large tables with little to no cycle penalties[5][4]. These predictors, while improving the overall prediction rate, do not address the branches which cannot be predicted well by the indexing scheme of the studied predictors (such as the branches described for the adpcm benchmark). As an alternative to larger tables, ignoring the need for longer history, some alternate research has focused on the reduction of contention via predictors such as the agree predictor or the bi-mode predictor. It is our belief that branch prediction research has nearly reached a plateau in which no further efforts into the development of a single branch predictor will prove fruitful, with one exception. We have shown that longer history registers in both local and global schemes provide better

correlation detection and thus prediction. Like the neural predictor, we believe that the next step in the evolution of generalized branch predictors will be one being able to use very long history registers without the need to double the PHT size for every history bit added.

Beyond the need for longer history, we believe that specific branches and algorithms should be studied in order to produce specialized branch predictors that are able to specifically target hard to predict branches. Two examples of such studies are the data correlating branch predictor[15] and the address-branch correlating predictor[14]. Following in the foot-steps of the tournament predictor[2] and the 2Bc-gskew[9], we will then be able to integrate such a predictor into a selection logic. Along with our support for specialized branch predictors, we have also created a new standard for the presentation of data as well as a few new statistics which aid in identifying good specialized branch predictors. First, we provide support for categorizing benchmarks according to the classification of branches they contain. Such an approach can allow a specialized predictor design to be tested on the appropriate benchmarks. In the case of our benchmarks, we have identified the adpcm benchmark as the one most worthy of research due to its very high miss rate. Second, we used analysis of the top 10 missed branches and the count of replaced and shifted branches to identify whether or not a different predictor performed better simply by predicting the same branches better or by predicting differently behaving branches more accurately. Lastly, we also used a more global version of the previous top 10 method in Section 4.4 where all the branches were placed into categories: *improved*, *constant*, or *degraded*. We then used these categories, specifically the dynamic reduction of mispredictions divided by the dynamic count of *improved* branches, to show conclusively why the local/gshare tournament predictor is clearly the correct configuration instead of bimodal/gshare or bimodal/local[1]. This path of development, unfortunately, will lead design down a familiar path of trading power and space for performance unless integrated with static schemes to allow for energy saving "hints" or very well tuned selection logic, most likely linked with low level cache meta data thus allowing some parts of the branch predictor to hibernate when not needed.

While the general focus of this research is geared towards uncovering a method for branch prediction development, most likely applicable in aggressive wide-issue and deeply

[1]We used the global classification even though in this case the simple top 10 approach would have worked. We believe that the global classification of all branches is crucial as the difference between predictors narrows and selection for a tournament style predictor needs to be made.

pipelined processors, the increasingly popular embedded processors command our attention. Old research with focus on delay slots has reached a dead end; other research, focusing on profiling data, seems to have its place in embedded processor development. In [11], Hwu et. al. describe a simple profiling scheme which is used to embed "hints" in the code generated by the compiler. These hints then help yield the best overall performance for the benchmarks when compared to other simple strategies used in the paper. Similarly, Fisher and Freudenberger[12] studied the use of a program trace to predict easily predicted branches which are also consistently found to be either taken or not-taken. In embedded applications, additional hardware costs may offset the benefits of the lowered miss rate described in the above, more complicated, predictors when compared to simple hardware and a higher miss rate. Due to the lower energy costs of smaller hardware and a willingness to spend more time in compiling and collecting profiling data, it appears that specialized static/dynamic collaborative methods in branch prediction for embedded applications may provide great benefits while maintaining the low power consumption. An example of such a configuration would be to use program execution traces to identify highly biased branches and statically predict them. Then, allow a specialized branch predictor to be designed for the hard to predict branches only. Such a design would allow for a "complex" predictor with less entries and moderate contention, since fewer branches require a prediction.

In either application, embedded or deeply pipelined super-scalar, the demand for better predictors can only increase as penalties for mispredictions become more costly in both cycles and power. Having apparently exhausted the ability to design a single general predictor with better average performance we find that both data and design must focus on individual benchmarks, algorithms, and individual branches. The future of branch prediction then is in development of specialized predictors, since specialized predictors have demonstrated potential in achieving better performance for specific branches such as those correlating with old data values or the address of data. Along with specialized branch predictors, advanced selection logic must be created. In past research, selection logic only needed to choose between two predictors, or in the case of the 2Bc-gskew a selection between a simple bimodal predictor and the average prediction provided from two tables by the e-gskew predictor. After reviewing the number of problem branches found in the adpcm benchmark alone, it becomes clear that for processors expecting to run a wide range of applications, a wide range of specialized branch predictors would have to be implemented. The development of

specialized prediction logic will end the trend of inefficiently utilized memory of very large predictors and perhaps even provide a means to reduce the large BTB. Thus, selection logic would then need to be able to select one or more (such as the average of a few) of many prediction results while being able to limit the access to predictors that will not contribute to the prediction during a give phase of execution.

REFERENCES

[1] Nathan L. Binkert et al., *The M5 Simulator: Modeling Networked Systems*. IEEE Micro, July/August 2006 (Vol. 26, No. 4) pp. 52-60. http://m5.eecs.umich.edu 1.1

[2] S. McFarling. *Combining Branch Predictors*. WRL Technical Note TN-36, June 1993. 3.2, 3.4, 5.2.2, 6

[3] T. Y. Yeh and Y. N. Patt. *Alternative implementations of two-level adaptive branch prediction*. In Proc. 19th Int. Sym. on Computer Architecture, pages 124-134, May 1992. 3.2, 3.3

[4] D. A. Jiménez. *Reconsidering Complex Branch Predictors*. In Proc. 9th Int. Sym. on HPCA, 2003. 5.1.4, 6

[5] Daniel A Jiménez, Stephen W. Keckler, and Calvin Lin. *The impact of delay on the design of branch predictors*. In Proceedings of the 33rd Annual Internation Symposium on Microarchitecture, pages 67-76, December 2000. 5.1.1, 5.1.2, 6

[6] J. E. Smith. *A study of branch prediction strategies*. In Proc. 8th Int. Sym. on Computer Architecture, pages 135-148, May 1981. 1

[7] T. Y. Yeh and Y. N. Patt. *A comparison of dynamic branch predictors that use two levels of branch history*. In Proc. 20th Int. Sym. on Computer Architecture, pages 124-134, May 1992. 1

[8] J. K. L. Lee and A. J. Smith. *Branch prediction strategies and branch target buffer design*. Computer, 17(1), January 1984. 1

[9] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakakis Sazeides. *Design tradeoffs for the Alpha EV8 conditional branch predictor*. In Proceedings of the 29th International Symposium on Computer Architecture, May 2002. 1, 6

[10] S. McFarling and J. Hennessy. *Reducing the cost of branches*. In Proc. 13th Int. Sym. on Computer Architecture, pages 396-403, June 1986. 6

[11] W. W. Hwu, T. M. Conte, and P. P. Chang. *Comparing software and hardware schemes for reducing the cost of branches*. In Proc. 16th Int. Sym. on Computer Architecture, pages 224-233, May 1989. 6

[12] J. A. Fisher and S. M. Freudenberger. *Predicting conditional branch directions from previous runs of a program.* In Proceedings of ASPLOS V, pages 85-95, Boston, MA, October 1992. 5.2.5, 6

[13] S. T. Pan, K. So, and J. T. Rahmeh. *Improving the accuracy of dynamic branch prediction using branch correlation.* In Proceedings of ASPLOS V, pages 76-84, Boston, MA, October 1992. 5.2.2

[14] Hongliang Gao, Yi, Ma, Martin Dimitrov, and Huiyang Zhou. *Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches.* In Proc. 14th Int. Sym. on HPCA, pages 74-85, 2008. 5.2.4, 6

[15] Timothy H. Heil, Zak Smith, and J. E. Smith. *Improving Branch Predictors by Correlating on Data Values.* In Proc. 5th Int. Sym. on HPCA, pages 28-37, 1999. 5.2.3, 6

[16] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. *The bi-mode branch predictor.* In Proceedings of the 30th Annual Internation Symposium on Microarchitecture, pages 4-13, December 1997. 5.2.1

[17] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. *The agree predictor: a mechanism for reducing negative branch history interference.* In Proceedings of the 24th International Symposium on Computer Architecture, June 1997. 5.2.5

[18] Daniel A Jiménez and Calvin Lin. *Dynamic Branch Prediction with Perceptrons.* In Proc. 7th Int. Sym. on HPCA, pages 197-206, 2001. 5.2.6

[19] Daniel A Jiménez and Calvin Lin. *Neural methods for dynamic branch prediction.* ACM Transactions on Computer Systems, 20(4), November 2002. 5.2.6

[20] Pierre Michaud, André Seznec, and Richard Uhlig. *Trading conflict and capacity aliasing in conditional branch predictors.* In Proceedings of the 24th International Symposium on Computer Architecture, June 1997.

[21] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. *Multiple-block ahead branch predictors.* In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 116-127, October 1996. 5.1.2

APPENDIX A

ALTERNATE REPRESENTATION OF BASE PREDICTOR RESULTS

Table A.1: Alternative representation of the **bimodal** predictor results.

Table Size	Normal Avg. Miss Rate	Deviant Avg. Miss Rate
1k	5.57%	26.30%
2k	5.37%	26.30%
4k	5.21%	26.30%
8k	5.18%	26.30%

Table A.2: Alternative representation of the **GAg** predictor results.

Table Size	Normal Avg. Miss Rate	Deviant Avg. Miss Rate
1k	6.84%	20.32%
2k	6.42%	19.92%
4k	6.23%	19.46%
8k	5.79%	19.18%
16k	5.51%	18.93%
32k	5.06%	18.73%
64k	4.61%	18.66%
128k	4.42%	18.61%

Table A.3: Alternative representation of the **gshare** predictor results.

Table Size	Normal Avg. Miss Rate	Deviant Avg. Miss Rate
1k	5.51%	21.13%
2k	4.87%	20.46%
4k	4.53%	20.03%
8k	4.44%	19.74%

Table A.4: Alternative representation of the **local** predictor results.

Lvl 1 Size	Lvl 2 size	Normal Avg. Miss Rate	Deviant Avg. Miss Rate	Lvl 1 Size	Lvl 2 size	Normal Avg. Miss Rate	Deviant Avg. Miss Rate
512	512	4.12%	18.79%	1k	512	3.92%	18.80%
512	1k	3.97%	18.78%	1k	1k	3.82%	18.80%
512	2k	3.92%	18.78%	1k	2k	3.77%	18.78%
512	4k	3.84%	18.79%	1k	4k	3.70%	18.78%
512	8k	3.81%	18.82%	1k	8k	3.66%	18.82%
2k	512	3.86%	18.80%	4k	512	3.83%	18.79%
2k	1k	3.75%	18.79%	4k	1k	3.71%	18.80%
2k	2k	3.70%	18.79%	4k	2k	3.67%	18.79%
2k	4k	3.62%	18.79%	4k	4k	3.60%	18.79%
2k	8k	3.59%	18.82%	4k	8k	3.57%	18.82%
8k	512	3.82%	18.79%				
8k	1k	3.70%	18.80%				
8k	2k	3.67%	18.79%				
8k	4k	3.59%	18.79%				
8k	8k	3.57%	18.82%				

APPENDIX B

DESTRUCTIVE INTERFERENCE

Destructive interference was first described along with the bimodal branch predictor in Section 3.1. It is the case where 2 or more branches map to the same entry and increase the miss rate beyond what that which would have resulted if they were to map to separate entries. Below is an example of such a branch from the g721 encode benchmark.

Table B.1: Table entries containing branch PC 120002de0 for g721 encode. *miss Rate* is for PC 120002de0 only.

	1k	2k	4k	8k
Table Index	888	888	2936	2936
num Accesses	294764	294763	147521	147519
num Misses	145691	145691	1049	1048
num PCs	4	3	2	1
num Switches	294479	294478	2	1
miss Rate*	99.26%	99.26%	0.71%	0.71%

This example was found in the *predictor_pole()* function at PC 120002de0. When executed using either a 1k or 2k bimodal predictor table we find that this particular branch has a miss rate of 99.26%; then, when executed with either a 3k or 4k bimodal predictor table we find a miss rate of 0.71% (more details can be seen in Table B.1). As can be seen in the table, PC 120002de0 shares a table index with 3 other PC in the 1k table: 120013de0, 120006de0, and 120004de0. When using a 2k table, PC 120013de0 is removed from entry 888 but only decreases the number of accesses to entry 888 by 1. This reveals that this branch was only executed once; further, we can see in Table B.1 that the number of misses in entry 888 was not reduced, meaning that the removed PC never missed. When switching to a 4k table

size we find that our branch PC 120002de0 moves to entry 2936 along with PC 120006de0 dropping 120004de0. Using the same table, we see that the number of accesses dropped by 50% while the number of misses dropped by 91% thus showing that PC 120004de0 was destructively interfering with our branch at PC 120002de0. Finally we see that dropping the final conflicting branch reduces the number of accesses by 2 and the number of misses by 1 meaning no real interference was caused by this PC.

APPENDIX C

CONSTRUCTIVE INTERFERENCE

Like destructive interference, constructive interference was discussed briefly along with the bimodal branch predictor. An instance of constructive interference is one where 2 or more branches map to the same entry and lower the miss rate when compared to the miss rate generated if these branches were to map to separate entries. The example below was taken from the g721 decode benchmark.

Table C.1: Entry 584 containing branch at PC 120008920 for g721 decode. *miss Rate* is for PC 120002920 only.

	1k	2k	4k	8k
num Accesses	442802	442553	442456	442456
num Misses	147537	147535	294936	294936
num PCs	3	2	1	1
num Switches	295280	295032	1	1
miss Rate*	50.01%	50.01%	99.99%	99.99%

When executing a branch within the function *_IO_new_file_xputn()* at address 120008920 we find a branch that yields a 50% miss rate when executed with a 1k or 2k bimodal predictor. That same branch yields a 99.99% miss rate when executed with either a 4k or 8k bimodal predictor. Via further analysis we find that this address indexes to entry number 584 in the prediction table for all the bimodal table sizes. This entry has the following characteristics presented in Table C.1. When looking at entry 584 in the 1k table we find 3 branch PCs: 120008920, 120002920, and 120001920. The value *num Switches* describes the total number of times this entry was accessed by a PC different from the last accessing PC. In this case we still have a difficult time determining which PC of the 3 is causing the conflict. When

looking at the data from the 2k table we see that PC 120001920 indexed into another entry (1608) while the other 2 remain on entry 584. Here we find that the number of switches only decreased by 248 while the miss rate stayed the same. This fact tells us that it was PC 120002920 that is causing the interference. When we move to the 4k and 8k tables we find that PC 120002920 indexed into entry 2632 and left our branch in question as the only PC using entry 584. At this point we also find that the miss rate nearly doubled; leaving us with the conclusion that PC 120002920 was causing constructive interference and helping reduce the miss rate.

APPENDIX D

BRANCH DEPENDENCY IN ADPCM CODER

```
/* adpcm_coder segment */
if(diff >= step){
  delta = 4;
  diff -= step;
  vpdiff += step;
}
step >>= 1;
if( diff >= step){
  delta |= 2;
  diff -= step;
  vpdiff += step;
}
step >>= 1;
if ( diff >= step ) {
  delta |= 1;
  vpdiff += step;
}
```

Figure D.1: Code from adpcm.c

Adpcm Coder code contains the code snippet (Figure D.1) in *void adpcm_coder(...)* which can be found in *adpcm.c*. The branch pattern is completely dependent on the data provided and further dependent on changes made at each iteration of the *for(...)* loop containing to code snippet. It is important to note from the code snippet that the data being used to make the branch decisions is also modified according to the path taken by the branches.

When using a simple bimodal predictor we find that the data dependency dominates the miss rate and forces a poor performance average in any table size. Similarly, the effects of the

Table D.1: Adpcm miss rates using the coder function

	Bimodal			
	1k	2k	4k	8k
Miss Rate:	28.49%	28.49%	28.49%	28.49%
	Gshare			
	1k	2k	4k	8k
Miss Rate:	24.75%	24.17%	23.35%	22.79%
	Local			
	512/512	1k/1k	2k/2k	4k/4k
Miss Rate:	21.52%	21.54%	21.52%	21.51%
	Bimodal/Gshare Tournament, 4-bit meta			
	8k/8k/1k	8k/8k/2k	8k/8k/4k	8k/8k/8k
Miss Rate:	26.54%	26.54%	26.54%	26.54%
	Local/Gshare Tournament, 4-bit meta			
	1k/1k/512	2k/2k/1k	4k/4k/2k	8k/8k/4k
Miss Rate:	21.76%	21.80%	21.76%	21.74%

data dependency dominate with the gshare and all other history based predictors. We found that the bimodal predictor out-performs all the other predictors for the 3 *if* statements in Figure D.1, except for in the cases of the 4k and 8k gshare. Such behavior was found to be caused by the lack of correlation between previous execution of the same or other branches and the future path. Instead, the more random predictions of the bimodal predictor were, by chance, more accurate but still statistically insignificant since the overall miss rate of the bimodal predictor was higher than that of other predictors.

Data was collected for some of the most missed branches in the adpcm coder benchmark and placed in Table D. Branches included for this study were the 3 *if* statements in Figure D.1, the initial comparison on input data *if(diff<0)*, the final *if(bufferstep)*, and the *for* loop containing these statements. It was then found that other branches in this working set are affected by the pseudo-random branch behavior of the 3 *if* statements when using global history based prediction. The prediction for the *for* loop performs worse as the size of the gshare predictor increases. Each time the *for* loop executes, several other branches with pseudo-random results execute. Such a behavior pollutes the history register and lengthen the warm-up time as it attempts to capture patterns for a simple branch such as the nearly

Table D.2: Adpcm in depth view of the miss rate for each of the top missed branches in the coder function

if(diff<0)	Bimodal	Gshare	Local		Tournament(Gshare/Local)	
1k	50.57%	44.71%	512/512	41.30%	1k/1k/512	41.84%
2k	50.57%	44.83%	1k/1k	41.45%	2k/2k/1k	42.00%
4k	50.57%	44.89%	2k/2k	41.43%	4k/4k/2k	41.97%
8k	50.57%	44.84%	4k/4k	41.55%	8k/8k/4k	42.14%
if(bufferstep)	Bimodal	Gshare	Local		Tournament(Gshare/Local)	
1k	50.00%	4.86%	512/512	0.00%	1k/1k/512	0.00%
2k	50.00%	5.51%	1k/1k	0.00%	2k/2k/1k	0.00%
4k	50.00%	5.62%	2k/2k	0.00%	4k/4k/2k	0.00%
8k	50.00%	4.72%	4k/4k	0.00%	8k/8k/4k	0.00%
for(...)	Bimodal	Gshare	Local		Tournament(Gshare/Local)	
1k	0.20%	5.88%	512/512	0.63%	1k/1k/512	0.62%
2k	0.20%	7.26%	1k/1k	0.57%	2k/2k/1k	0.57%
4k	0.20%	7.53%	2k/2k	0.55%	4k/4k/2k	0.55%
8k	0.20%	7.86%	4k/4k	0.56%	8k/8k/4k	0.56%
Triple if(...)	Bimodal	Gshare	Local		Tournament(Gshare/Local)	
1k	37.73%	39.00%	512/512	39.97%	1k/1k/512	40.41%
2k	37.73%	38.03%	1k/1k	39.98%	2k/2k/1k	40.44%
4k	37.73%	37.27%	2k/2k	39.94%	4k/4k/2k	40.37%
8k	37.73%	36.88%	4k/4k	39.87%	8k/8k/4k	40.26%

always taken backwards branch of the *for* loop. As the history register increases in size, more random bits are inserted at a disproportionate rate when compared to the relevant one of the *for* loop. It then takes a considerably larger number of cycles for the gshare predictor to execute and update with all the possible permutations of the random bits so that the prediction is consistent for every instance of the branch. Alternatively, the local predictor is able to predict this branch with a only a few mispredictions. While all these branches perform slightly worse, the prediction for the statement *if(bufferstep)*, having the pattern TNTNTNTNTN..., is improved drastically. Compared to the bimodal predictor, the better prediction of simple pattern branches, such as *if(bufferstep)* by the gshare predictor greatly offset the slightly worse prediction of the *for* loop caused by added pollution to the history register, as can be seen in the overall miss rates provided in Table D. Similarly, the local predictor is able to further improve upon the already low miss rate of the gshare predictor

for this branch, and reduce the miss rate to 0% (the raw numbers show roughly 10 misses allowing for the warm-up of the local history register).

In Section 3.4.1, *bimodal/gshare tournament predictor*, it was said that the cause of the poor performance of the tournament predictor was the changing of phases by the meta predictor caused by pseudo-random branch behavior. A sample of the actual, as well as the predicted, branch behavior was taken from each of the predictors for the 1k/1k/1k configuration (a sample of which is presented in Table D).

Table D.3: Differences in prediction pattern of a data driven branch in adpcm's coder function

	Taken/Not taken pattern
Mod 10	1---5----10----5----20----5----30----5----40----5----
Actual:	NNNTTNNTTN NNTNTNNTTN TNTNNTNNNT TNNTNNTTNN TNNTNTTNT
Bimodal:	NNNNNTNNNT NNNNNNNNNT NTNTNNNNNN NTNNNNNNTN NNNNNNNTN
Gshare:	NNTTTNNTNN NTTNNNNTNN TNTNNNNNNT NNTTNNTNNT NNTTNTTNN
Tournament:	NNNNNNNTNN NTTNNNNTNN TNTNNNNNNT NNTTNTTNNT NNTNNTTNN

In Table D we can spot one case of a bad phase change that is caused by the low number of states in the 2 bit meta predictor. The example starts in the bimodal phase (meta state 01) and quickly, on branch 6, has enough hits on the gshare predictor to switch states. It will then remain in the gshare phase until branch 40 has a miss while the bimodal predictor was correct. The state for the branch then shifts to *Weakly Gshare*. In execution 41 and 42 of the branch both predictors agree so no update is made to the meta predictor. Then, the bimodal predictor is correct again in execution 43 and the branch now resides in the *Weakly Bimodal* state. Finally, when execution 44 uses the bimodal predictor it misses and must shift back to the *Weakly Gshare* state. We can see in this example of 49 executions of this one particular branch an instance where a phase shift caused by the pseudo-random pattern of branch behavior causes a needless miss. While this may not seem like much, this particular branch executes a total of 684,432 times and has a miss rate of 47.43%. Further, if we could rid the predictor of phase shifts such as the one described above, we could reduce the miss rate for this particular branch by roughly 3% which almost entirely accounts for the difference in miss rate between the bimodal/gshare tournament predictor and gshare predictor alone.

BIOGRAPHICAL SKETCH

Yuval Peress

Yuval Peress was born on May 8th of 1984 in Ramat Hasharon, Israel. In the Sprint of 2005 he completed his Bachelors of Science degree in Computer Science at Florida State University. He then enrolled for his masters and doctoral degree in the fall of 2005.

Yuval's interests include computer architecture, mechanical and electrical engineering, and teaching.

Yuval lives in Tallahassee, Florida.